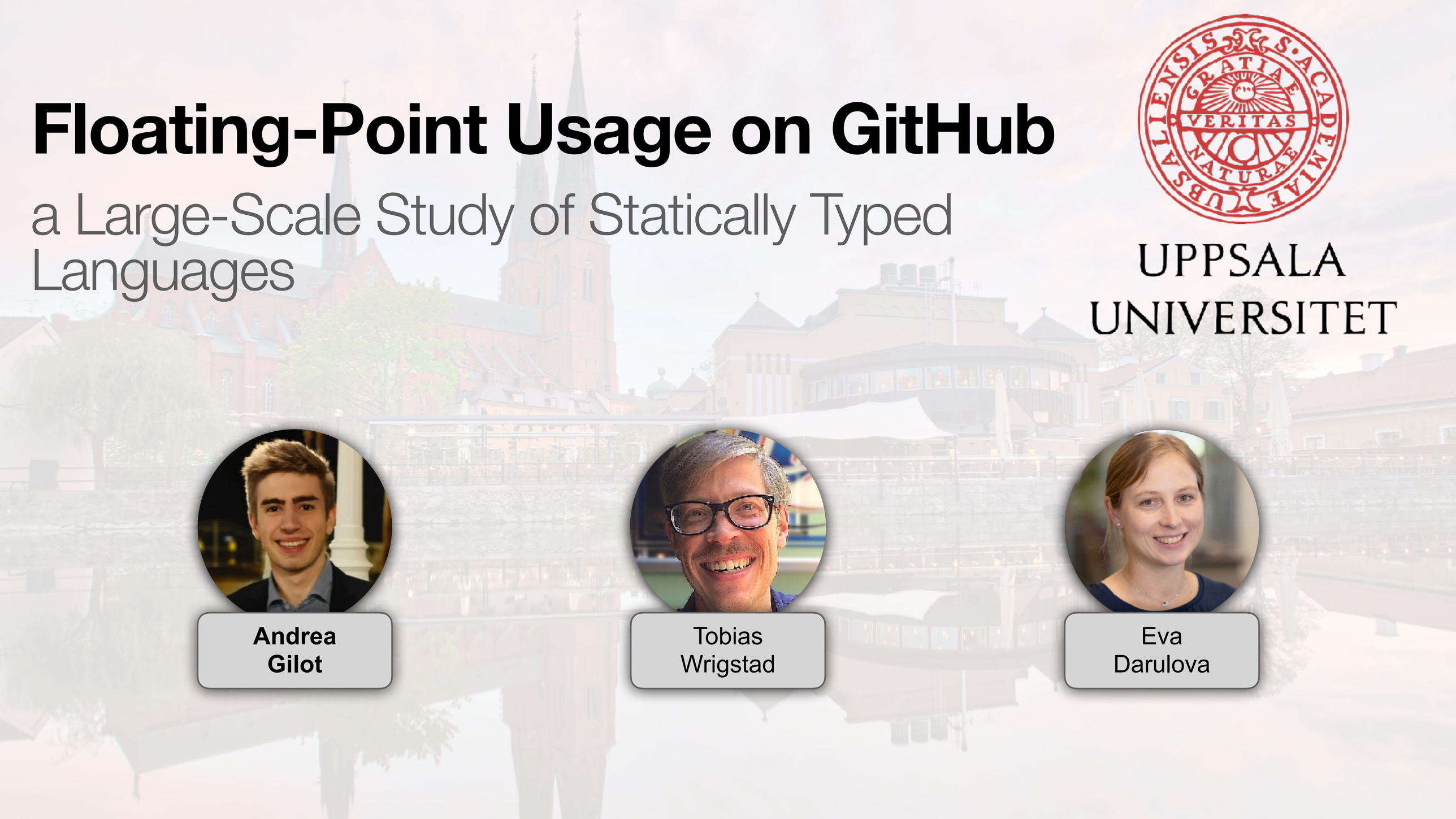
Floating-Point Usage on GitHub

a Large-Scale Study of Statically Typed Languages



UPPSALA UNIVERSITET

Andrea Gilot

Tobias Wrigstad

Eva Darulova

Recipe for a floating-point analysis paper

Recipe for a floating-point analysis paper

"Floating-point numbers are **often** used as a finite representation of real numbers in computer programs."

An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs, Titolo et al., VMCAI 2018

"**Many** algorithms are conceived (and even formally verified) in real numbers, but ultimately deployed using floating-point numbers"

Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions, Soloyev et al., TOPLAS 2018

Recipe for a floating-point analysis paper

"Floating-point numbers are **often** used as a finite representation of real numbers in computer programs."

$$\gamma_{\dot{\Pi}}(\dot{C}) := \bigsqcup\{\langle\gamma_{\mathbb{B}}(\dot{\eta})\rangle_t \twoheadrightarrow (r, \gamma_{\mathbb{E}}(\dot{e}))^\pi \mid \exists\langle\dot{\eta}\rangle_t \twoheadrightarrow (R, \dot{e})^\pi \in \dot{C},\ r \in R\}$$

An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs, Titolo et al., VMCAI 2018

"**Many** algorithms are conceived (and even formally verified) in real numbers, but ultimately deployed using floating-point numbers"

$$\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) = \tilde{f}(\mathbf{x}, \mathbf{0}, \mathbf{0}) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) e_i + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$$

Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions, Soloyev et al., TOPLAS 2018

Recipe for a floating-point analysis paper

"Floating-point numbers are **often** used as a finite representation of real numbers in computer programs."

$$\gamma_{\dot{H}}(\dot{C}) := \bigsqcup\{\langle\gamma_{\mathbb{B}}(\dot{\eta})\rangle_t \twoheadrightarrow (r, \gamma_{\mathbb{E}}(\dot{e}))^\pi \mid \exists\langle\dot{\eta}\rangle_t \twoheadrightarrow (R, \dot{e})^\pi \in \dot{C},\ r \in R\}$$

kepler0	1.09E-13	1.03E-13
kepler1	4.68E-13	*3.51E-13*
kepler2	2.38E-12	2.24E-12
predatorPrey	*1.67E-16*	2.35E-16
rigidBody1	**2.95E-13**	*3.22E-13*
rigidBody2	*3.61E-11*	3.65E-11
sine	6.91E-16	7.41E-16
sineOrder3	*6.54E-16*	1.09E-15
sphere	n/a	n/a
sqroot	5.35E-16	6.83E-16
t_div_t1	9.99E+00	2.80E-12
turbine1	2.41E-14	3.09E-14
turbine2	3.32E-14	*2.59E-14*
turbine3	3.52E-01	1.34E-14
verhulst	*2.84E-16*	4.80E-16

An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs, Titolo et al., VMCAI 2018

"**Many** algorithms are conceived (and even formally verified) in real numbers, but ultimately deployed using floating-point numbers"

$$\tilde{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) = \tilde{f}(\mathbf{x}, \mathbf{0}, \mathbf{0}) + \sum_{i=1}^{k} \frac{\partial \tilde{f}}{\partial e_i}(\mathbf{x}, \mathbf{0}, \mathbf{0}) e_i + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})$$

kepler0	6	15	0	7.5e-14	5.3e-14
kepler1	4	24	0	2.9e-13	1.6e-13
kepler2	6	36	0	1.6e-12	8.4e-13
predPrey	1	7	0	1.6e-16	1.5e-16
rigidBody1	3	7	0	3.0e-13	2.7e-13
rigidBody2	3	14	0	3.7e-11	3.0e-11
turbine1	3	14	0	1.7e-14	1.1e-14
turbine2	3	10	0	2.0e-14	1.4e-14
turbine3	3	14	0	9.6e-15	6.2e-15
verhulst	1	4	0	2.5e-16	2.4e-16

Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions, Soloyev et al., TOPLAS 2018

Recipe for a floating-point analysis paper

kepler0	1.09E-13	1.03E-13
kepler1	4.68E-13	*3.51E-13*
kepler2	2.38E-12	2.24E-12
predatorPrey	*1.67E-16*	2.35E-16
rigidBody1	**2.95E-13**	*3.22E-13*
rigidBody2	*3.61E-11*	3.65E-11
sine	6.91E-16	7.41E-16
sineOrder3	*6.54E-16*	1.09E-15
sphere	n/a	n/a
sqroot	5.35E-16	6.83E-16
t_div_t1	9.99E+00	2.80E-12
turbine1	2.41E-14	3.09E-14
turbine2	3.32E-14	*2.59E-14*
turbine3	3.52E-01	1.34E-14
verhulst	*2.84E-16*	4.80E-16

Recipe for a floating-point analysis paper

kepler0	1.09E-13	1.03E-13
kepler1	4.68E-13	*3.51E-13*
kepler2	2.38E-12	2.24E-12
predatorPrey	*1.67E-16*	2.35E-16
rigidBody1	**2.95E-13**	*3.22E-13*
rigidBody2	*3.61E-11*	3.65E-11
sine	6.91E-16	7.41E-16
sineOrder3	*6.54E-16*	1.09E-15
sphere	n/a	n/a
sqroot	5.35E-16	6.83E-16
t_div_t1	9.99E+00	2.80E-12
turbine1	2.41E-14	3.09E-14
turbine2	3.32E-14	*2.59E-14*
turbine3	3.52E-01	1.34E-14
verhulst	*2.84E-16*	4.80E-16

```
//@ requires 0.1 <= x <= 0.3;
double predatorPrey(double x) {
    double r = 4.0;
    double K = 1.11;
    return ((r * x) * x) / (1.0 + ((x / K) * (x / K)));
}
```

```
//@ requires -10 <= x <= 10 && 0 <= r <= 10 &&
-1.570796 <= lat <= 1.570796 && -3.14159265 <= lon <= 3.14159265;
double sphere(double x, double r, double lat, double lon) {
    double sinLat = sin(lat);
    double cosLon = cos(lon);
    return x + ((r * sinLat) * cosLon);
}
```

```
//@ requires -4.5 <= v <= -0.3 && 0.4 <= w <= 0.9 && 3.8 <= r <= 7.8;
double turbine1(double v, double w, double r) {
    return ((3.0 + (2.0 / (r * r))) - (((0.125 * (3.0 - (2.0 * v))) *
           (((w * w) * r) * r)) / (1.0 - v)))- 4.5;
}
```

Recipe for a floating-point analysis paper



FPBench

kepler0	1.09E-13	1.03E-13
kepler1	4.68E-13	*3.51E-13*
kepler2	2.38E-12	2.24E-12
predatorPrey	*1.67E-16*	2.35E-16
rigidBody1	**2.95E-13**	*3.22E-13*
rigidBody2	*3.61E-11*	3.65E-11
sine	6.91E-16	7.41E-16
sineOrder3	*6.54E-16*	1.09E-15
sphere	n/a	n/a
sqroot	5.35E-16	6.83E-16
t_div_t1	9.99E+00	2.80E-12
turbine1	2.41E-14	3.09E-14
turbine2	3.32E-14	*2.59E-14*
turbine3	3.52E-01	1.34E-14
verhulst	*2.84E-16*	4.80E-16

```c
//@ requires 0.1 <= x <= 0.3;
double predatorPrey(double x) {
    double r = 4.0;
    double K = 1.11;
    return ((r * x) * x) / (1.0 + ((x / K) * (x / K)));
}
```

```c
//@ requires -10 <= x <= 10 && 0 <= r <= 10 &&
-1.570796 <= lat <= 1.570796 && -3.14159265 <= lon <= 3.14159265;
double sphere(double x, double r, double lat, double lon) {
    double sinLat = sin(lat);
    double cosLon = cos(lon);
    return x + ((r * sinLat) * cosLon);
}
```

```c
//@ requires -4.5 <= v <= -0.3 && 0.4 <= w <= 0.9 && 3.8 <= r <= 7.8;
double turbine1(double v, double w, double r) {
    return ((3.0 + (2.0 / (r * r))) - (((0.125 * (3.0 - (2.0 * v))) *
           (((w * w) * r) * r)) / (1.0 - v)))- 4.5;
}
```

Toward a Standard Benchmark Format and Suite for Floating-Point Analysis, Damouche et al., NSV 2016

What do representative FP functions look like?

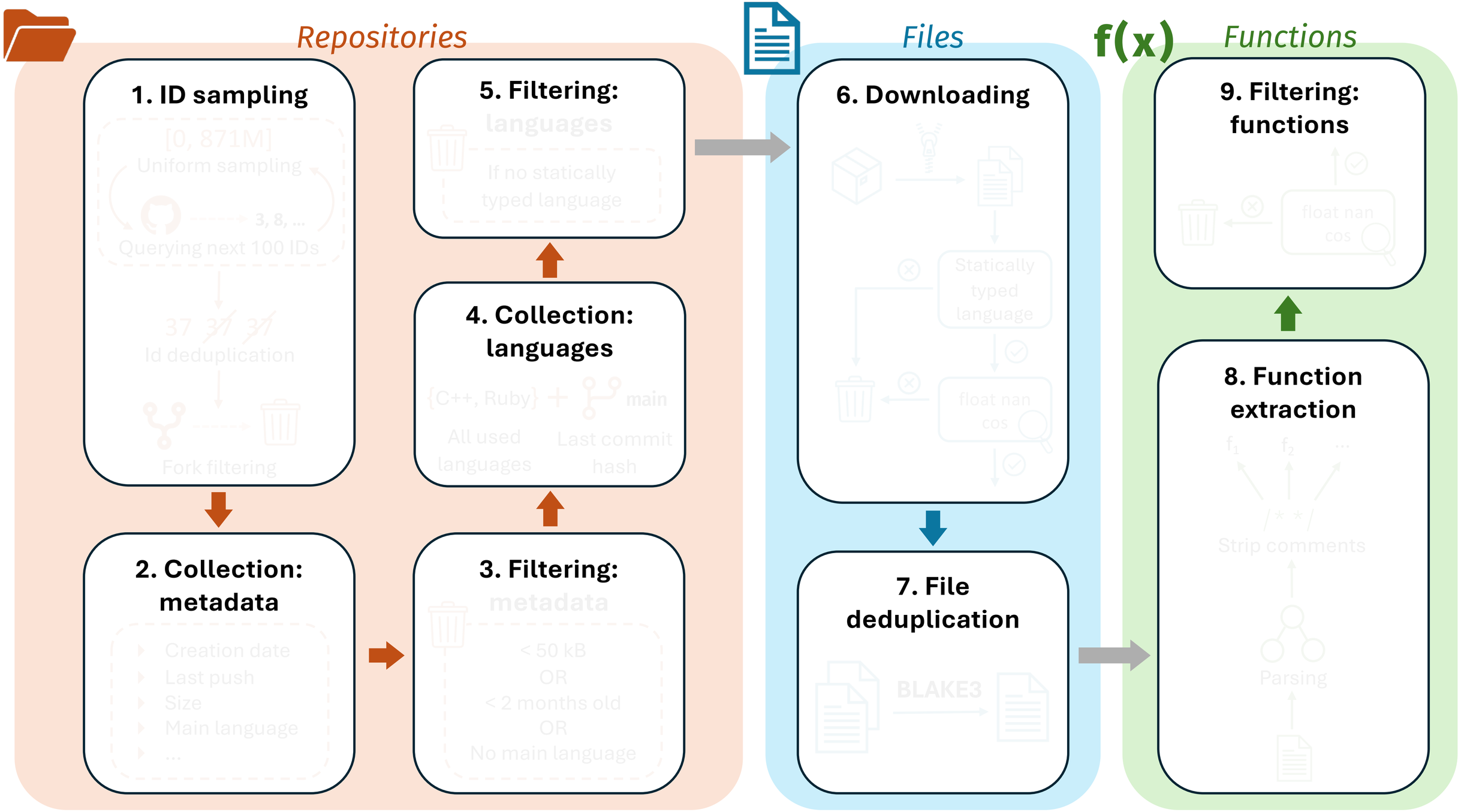
How common is FP code actually?

What do representative FP functions look like?

How common is FP code actually?



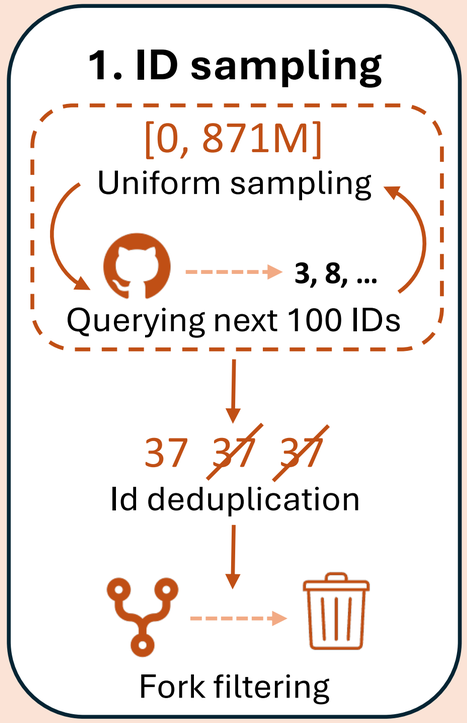
What do representative FP functions look like?


Repositories
Files
f(x)
Functions
1. ID sampling
[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37
Id deduplication
Fork filtering
2. Collection: metadata
Creation date
Last push
Size
Main language
...
3. Filtering: metadata
< 50 kB
OR
< 2 months old
OR
No main language
4. Collection: languages
{C++, Ruby}
main
All used languages
Last commit hash
5. Filtering: languages
If no statically typed language
6. Downloading
Statically typed language
float nan cos
7. File deduplication
BLAKE3
8. Function extraction
Strip comments
Parsing
9. Filtering: functions
float nan cos

Repositories

Files

Functions



Repositories

Files

Functions

1. ID sampling

[0, 871M]

Uniform sampling

3, 8, ...

Querying next 100 IDs

37 ~~37~~ ~~37~~

Id deduplication

Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

Repositories

Files

f(x) Functions

1. ID sampling

[0, 871M]
Uniform sampling

3, 8, ...

Querying next 100 IDs

37 ~~37~~ ~~37~~
Id deduplication

Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

Repositories

1. ID sampling

[0, 871M]
Uniform sampling

3, 8, ...
Querying next 100 IDs

37 ~~37~~ ~~37~~
Id deduplication

Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments, Maj et al., ECOOP 2024

Filtering is necessary, but using **intrinsic** properties (i.e. not stars)

Repositories

1. ID sampling

[0, 871M]
Uniform sampling

3, 8, …
Querying next 100 IDs

37 ~~37~~ ~~37~~
Id deduplication

Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- …

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments, Maj et al., ECOOP 2024

Filtering is necessary, but using **intrinsic** properties (i.e. not stars)

What we found:

Repositories

1. ID sampling

[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37 ~~37~~ ~~37~~
Id deduplication
Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

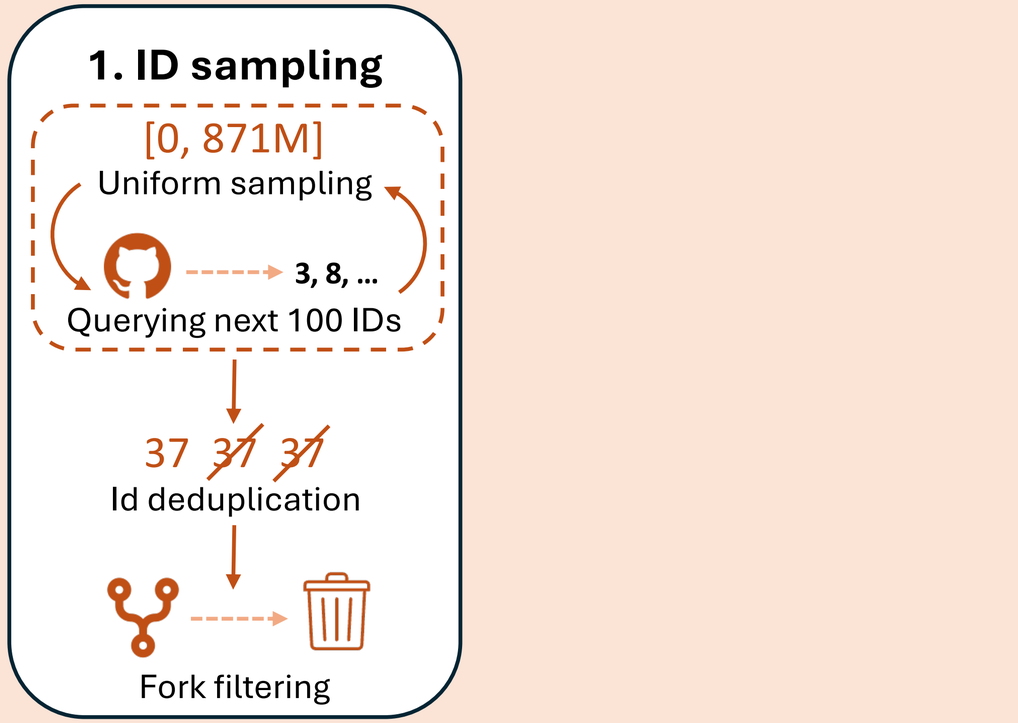
The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments, Maj et al., ECOOP 2024

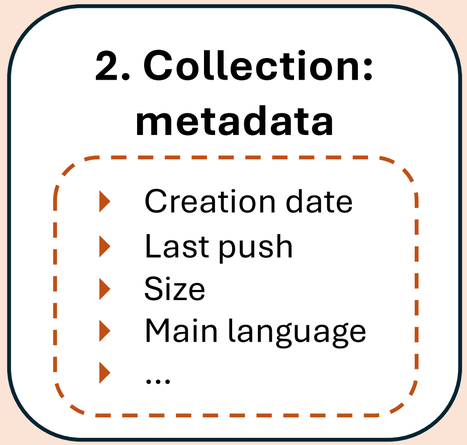
Filtering is necessary, but using **intrinsic** properties (i.e. not stars)

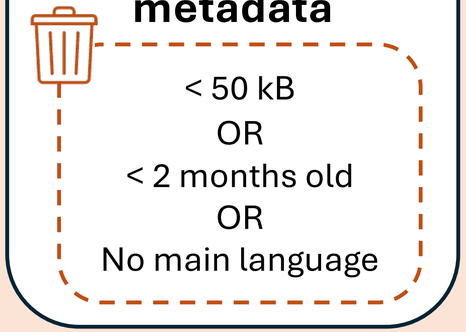
What we found:

19% empty repos

Repositories







The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments, Maj et al., ECOOP 2024

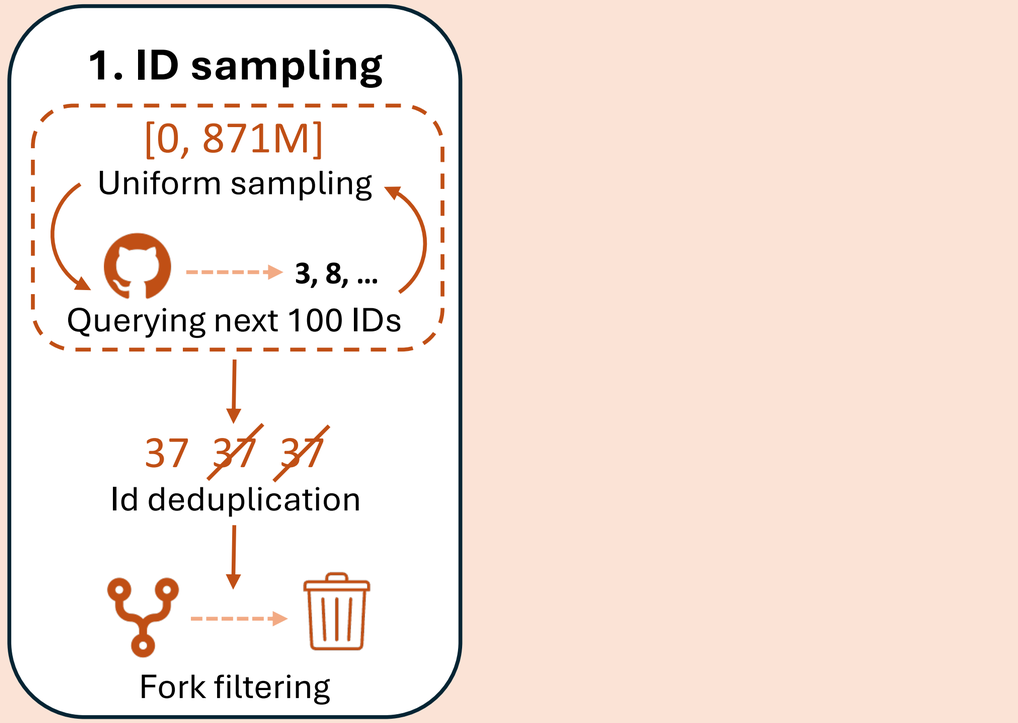
Filtering is necessary, but using **intrinsic** properties (i.e. not stars)

What we found:

19% empty repos

12% non-empty but without code

Repositories



The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments, Maj et al., ECOOP 2024

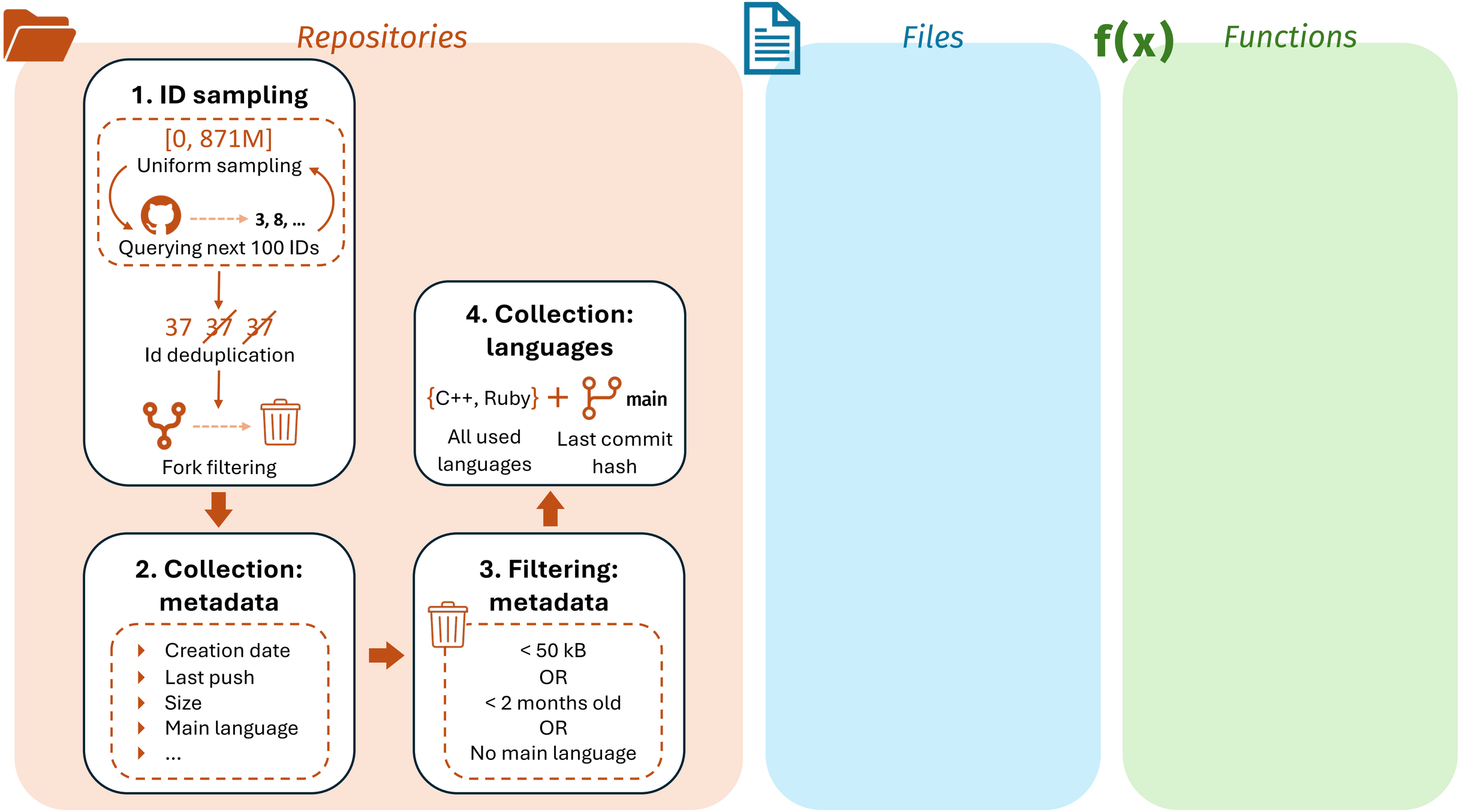
Filtering is necessary, but using **intrinsic** properties (i.e. not stars)

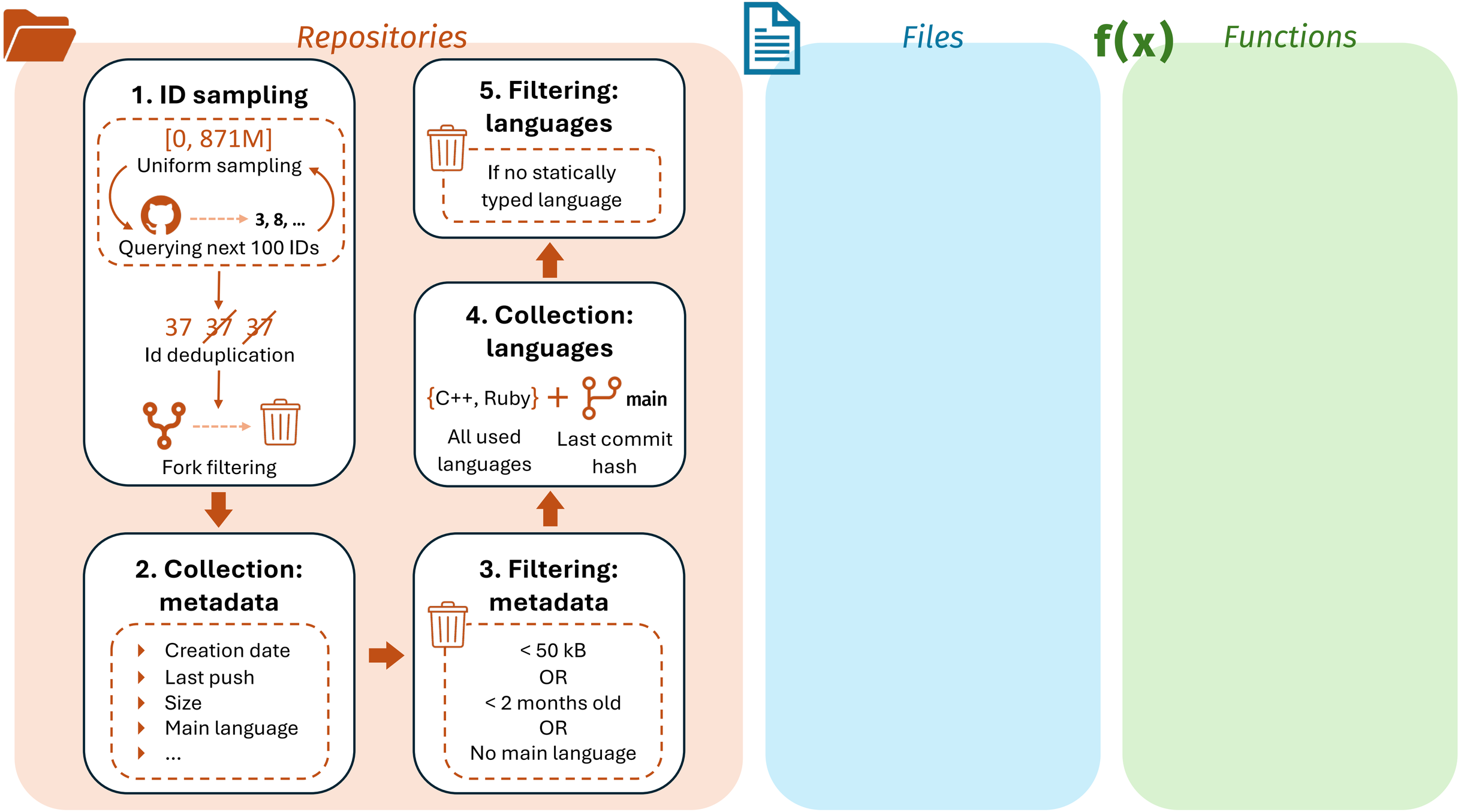
What we found:

19% empty repos

12% non-empty but without code

81% less than 2 months of activity


Repositories
Files
f(x)
Functions
1. ID sampling
[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37 37 37
Id deduplication
Fork filtering
2. Collection: metadata
Creation date
Last push
Size
Main language
...
3. Filtering: metadata
< 50 kB
OR
< 2 months old
OR
No main language
4. Collection: languages
{C++, Ruby} + main
All used languages
Last commit hash


Repositories
Files
f(x)
Functions
1. ID sampling
[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37 37 37
Id deduplication
Fork filtering
2. Collection: metadata
Creation date
Last push
Size
Main language
...
3. Filtering: metadata
< 50 kB
OR
< 2 months old
OR
No main language
4. Collection: languages
{C++, Ruby} + main
All used languages
Last commit hash
5. Filtering: languages
If no statically typed language

Repositories

Files

f(x) Functions

1. ID sampling

[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37 ~~37~~ ~~37~~
Id deduplication
Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

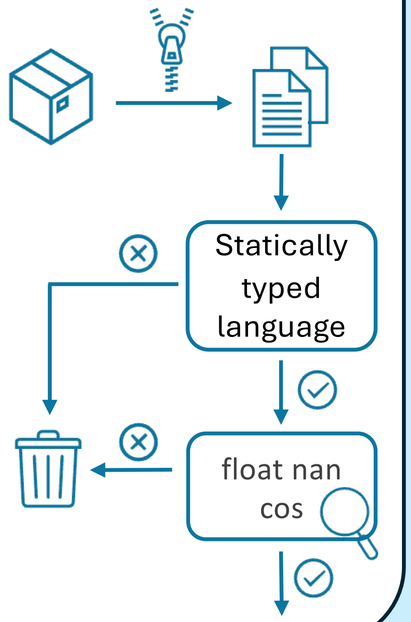
4. Collection: languages

{C++, Ruby} + main
All used languages
Last commit hash

5. Filtering: languages

If no statically typed language

6. Downloading



Repositories

Files

f(x) Functions

1. ID sampling

[0, 871M]
Uniform sampling

3, 8, ...

Querying next 100 IDs

37 ~~37~~ ~~37~~

Id deduplication

Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

4. Collection: languages

{C++, Ruby} + main

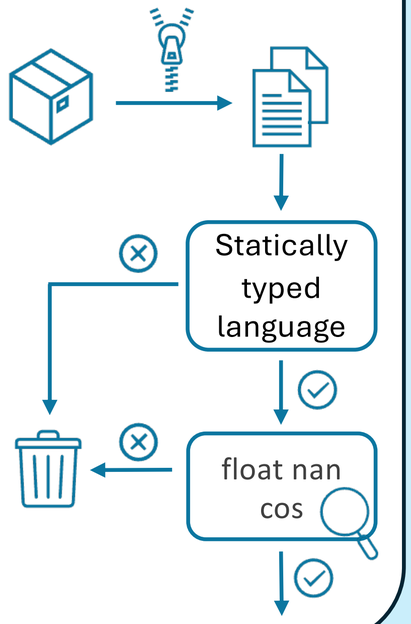
All used languages

Last commit hash

5. Filtering: languages

If no statically typed language

6. Downloading




```
17 double square(double x) {
18   return x * x;
19 }
```

Repositories

Files

f(x) Functions

1. ID sampling

[0, 871M]
Uniform sampling

3, 8, ...

Querying next 100 IDs

37 ~~37~~ ~~37~~

Id deduplication

Fork filtering

2. Collection: metadata

- Creation date
- Last push
- Size
- Main language
- ...

3. Filtering: metadata

< 50 kB
OR
< 2 months old
OR
No main language

4. Collection: languages

{C++, Ruby} + main

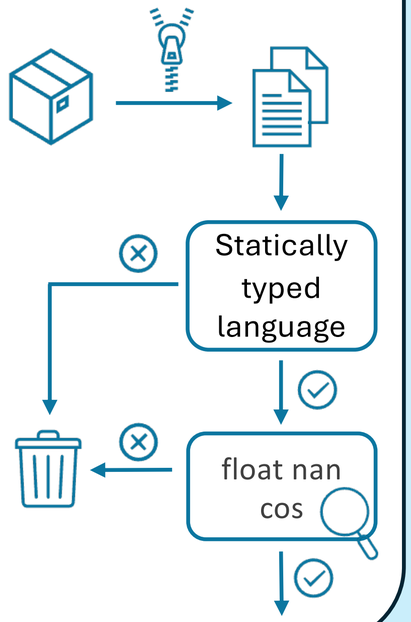
All used languages

Last commit hash

5. Filtering: languages

If no statically typed language

6. Downloading



7. File deduplication



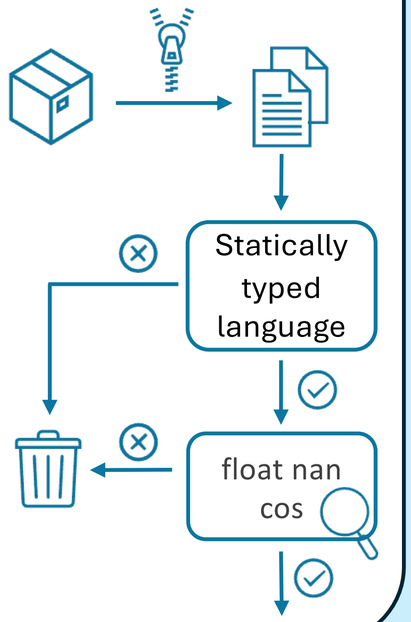
DéjàVu: A Map of Code Duplicates on GitHub, Lopes et al., OOPSLA 2017

"70% of the code on GitHub consists of clones of previously created files"

Files

Functions

6. Downloading



7. File deduplication



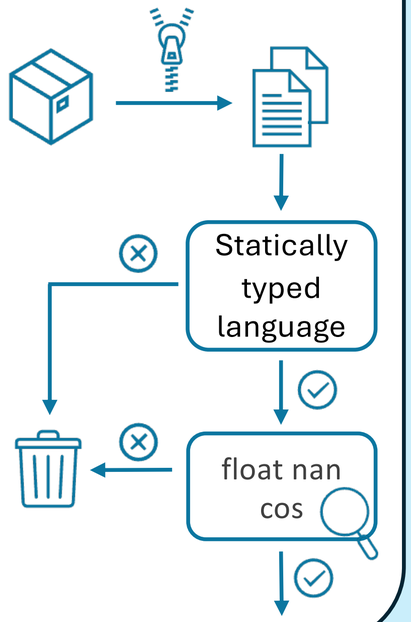
DéjàVu: A Map of Code Duplicates on GitHub, Lopes et al., OOPSLA 2017

"70% of the code on GitHub consists of clones of previously created files”

What we found:

Files

6. Downloading



7. File deduplication



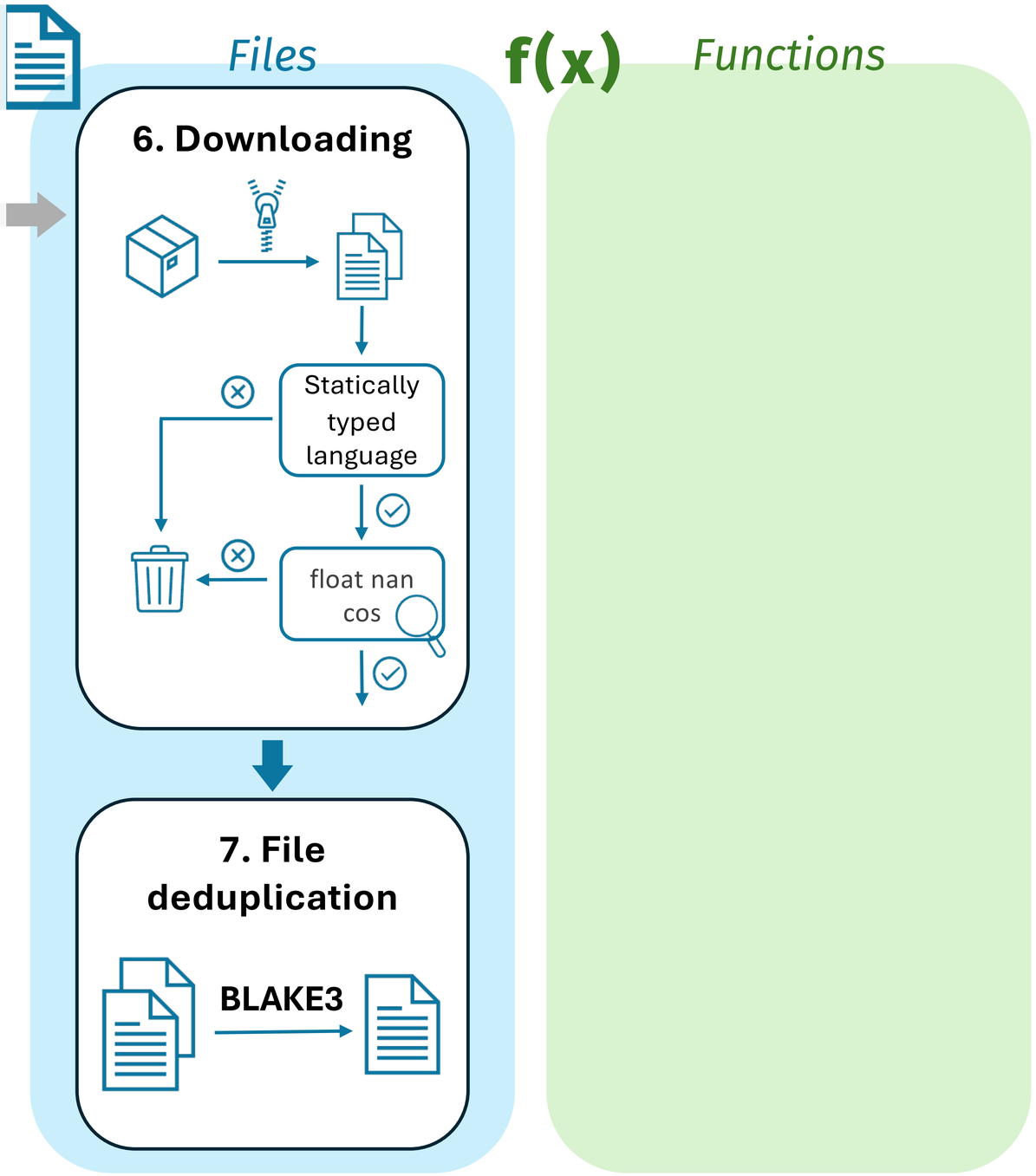
f(x) Functions

DéjàVu: A Map of Code Duplicates on GitHub, Lopes et al., OOPSLA 2017

"70% of the code on GitHub consists of clones of previously created files”

What we found:

63% of the files were duplicates



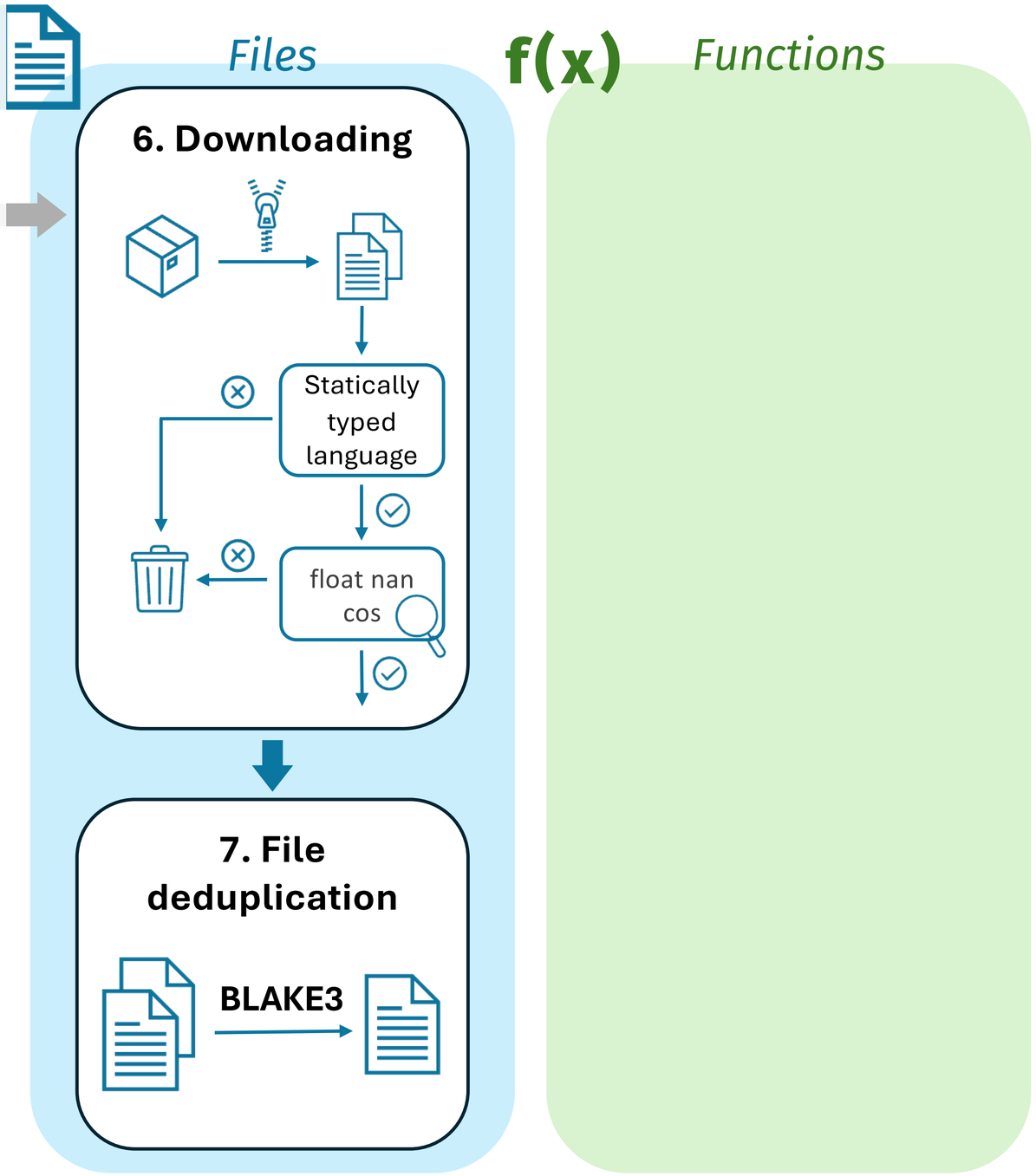
DéjàVu: A Map of Code Duplicates on GitHub, Lopes et al., OOPSLA 2017

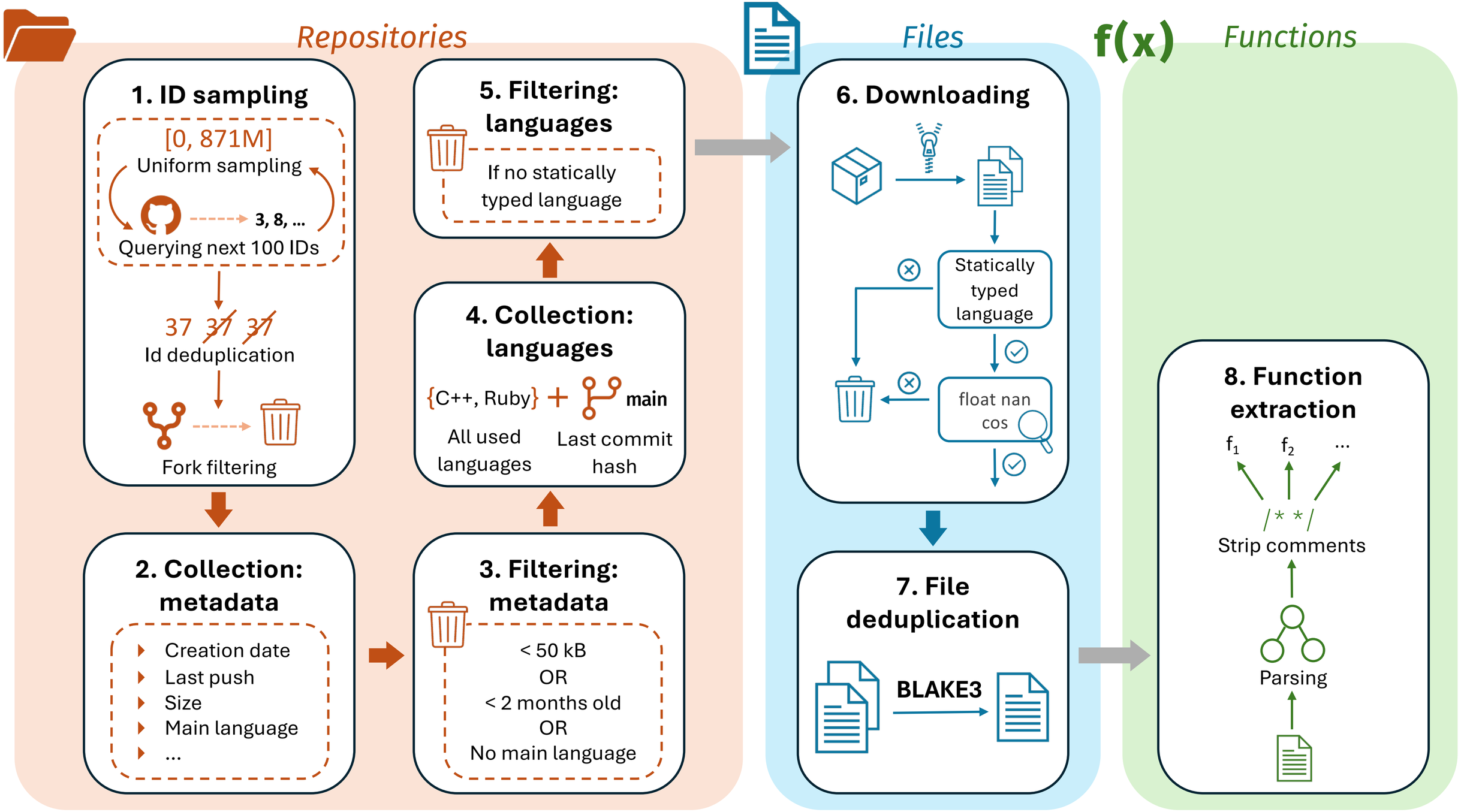
"70% of the code on GitHub consists of clones of previously created files”

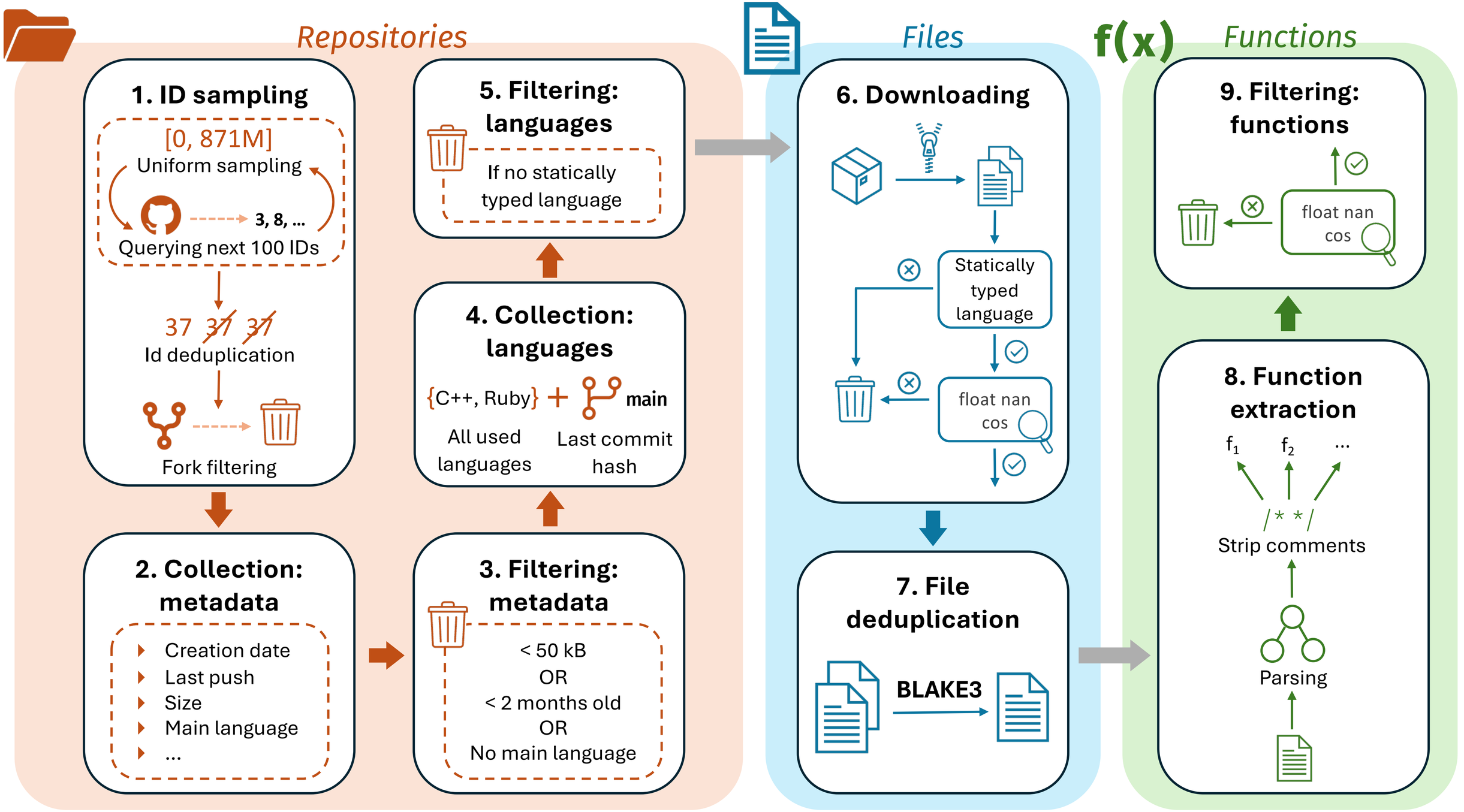
What we found:

63% of the files were duplicates

Some appeared 27 000 times



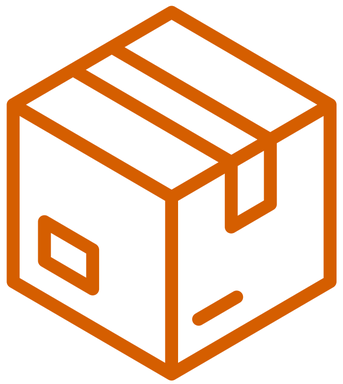

Repositories
Files
f(x)
Functions
1. ID sampling
[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37 37 37
Id deduplication
Fork filtering
2. Collection: metadata
Creation date
Last push
Size
Main language
...
3. Filtering: metadata
< 50 kB
OR
< 2 months old
OR
No main language
4. Collection: languages
{C++, Ruby} + main
All used languages
Last commit hash
5. Filtering: languages
If no statically typed language
6. Downloading
Statically typed language
float nan cos
7. File deduplication
BLAKE3
8. Function extraction
f1 f2 ...
/* */
Strip comments
Parsing


Repositories
Files
f(x)
Functions
1. ID sampling
[0, 871M]
Uniform sampling
3, 8, ...
Querying next 100 IDs
37 37 37
Id deduplication
Fork filtering
2. Collection: metadata
Creation date
Last push
Size
Main language
...
3. Filtering: metadata
< 50 kB
OR
< 2 months old
OR
No main language
4. Collection: languages
{C++, Ruby} + main
All used languages
Last commit hash
5. Filtering: languages
If no statically typed language
6. Downloading
Statically typed language
float nan cos
7. File deduplication
BLAKE3
8. Function extraction
f1 f2 ...
/* */
Strip comments
Parsing
9. Filtering: functions
float nan cos

Final floating-point dataset

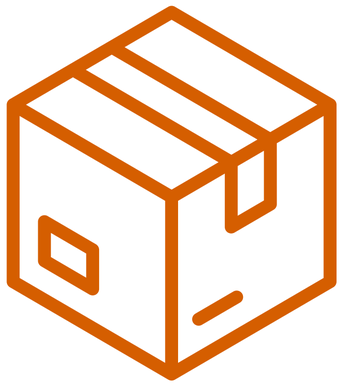
Final floating-point dataset

300k repositories



Final floating-point dataset

300k repositories

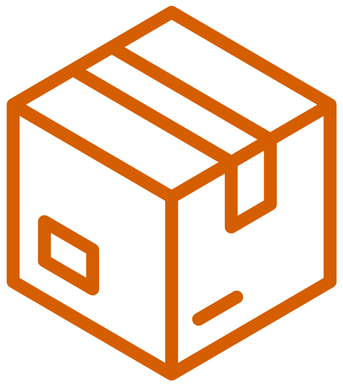


5M unique files



Final floating-point dataset

300k repositories



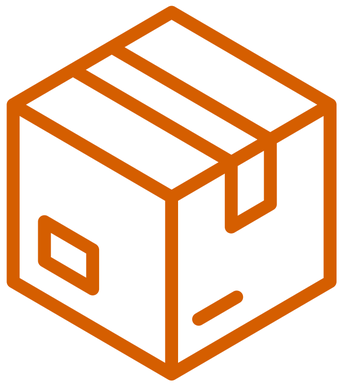
5M unique files



10M functions

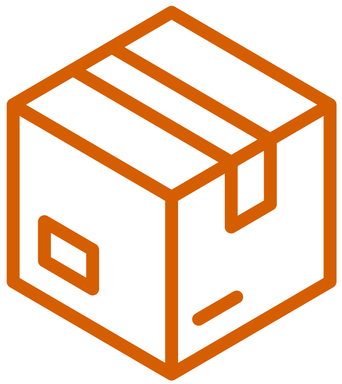
f(x)

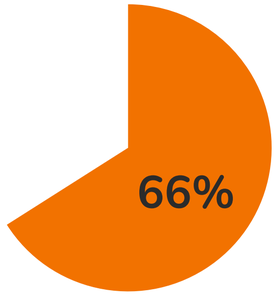
300k repositories



300k repositories

out of 465k we mined

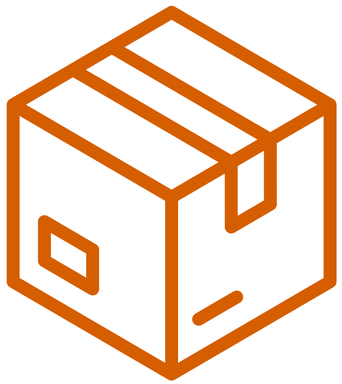




contain floating-point related **terms**

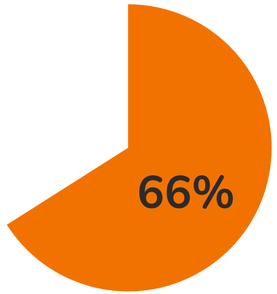
300k repositories

out of 465k we mined



```scala
def double(x: Int): Int = x * 2
```

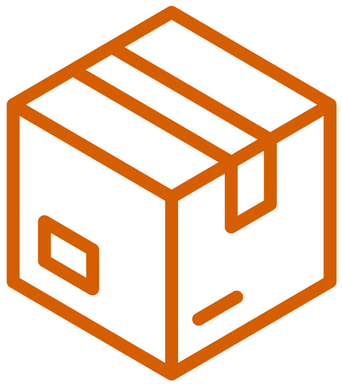
Example of false positive



contain floating-point related **terms**

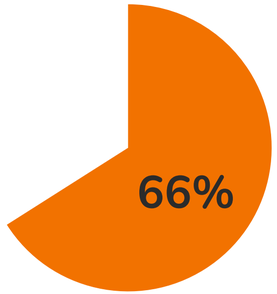
300k repositories

out of 465k we mined



```
def double(x: Int): Int = x * 2
```

Example of false positive



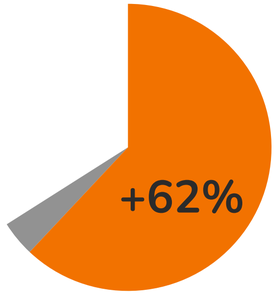
contain floating-point related **terms**

300k repositories

out of 465k we mined



Manual inspection on a sample

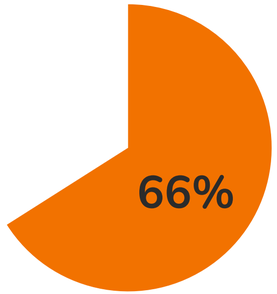


contain floating-point **code**

(with 95% confidence)

```
def double(x: Int): Int = x * 2
```

Example of false positive



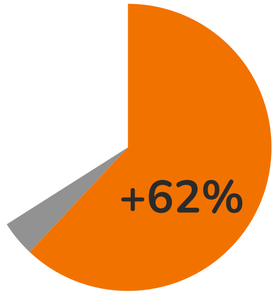
contain floating-point related **terms**

300k repositories

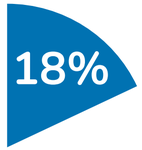
out of 465k we mined



Manual inspection on a sample



contain floating-point **code**
(with 95% confidence)



of **files** contain floating-point related **terms**
in the median repository

How common is FP code actually?

What do representative FP functions look like?

How common is FP code actually?

What do representative FP functions look like?

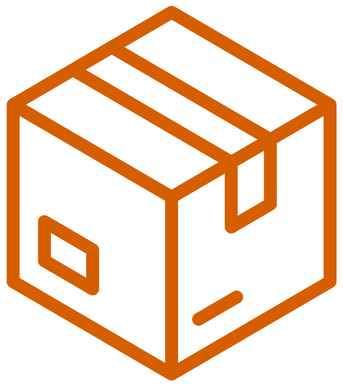
How common is FP code actually?

Most GitHub repos in our dataset (> 62%) contain floating-point code, but usually in only a small share of files.

What do representative FP functions look like?

Final floating-point dataset

300k repositories



5M unique files



10M functions

f(x)

Final floating-point dataset

300k repositories

5M unique files

10M functions

f(x)

f(x)

f(x)

- C
- C++
- C#
- Go
- Java
- TS

Floating-point functions statistics on GitHub

f(x)	# words	# params
C	61	3
C++	42	2
C#	34	2
Go	42	2
Java	28	1
TS	27	1

Floating-point functions statistics on GitHub

f(x)	# words	# params	Conditionals	
			Presence	# if present
C	61	3	57%	3
C++	42	2	54%	3
C#	34	2	42%	2
Go	42	2	63%	2
Java	28	1	37%	2
TS	27	1	44%	2

Floating-point functions statistics on GitHub

			Conditionals		Function calls
f(x)	# words	# params	Presence	# if present	
C	61	3	57%	3	
C++	42	2	54%	3	
C#	34	2	42%	2	
Go	42	2	63%	2	
Java	28	1	37%	2	
TS	27	1	44%	2	

Floating-point functions statistics on GitHub

			Conditionals		Function calls	
f(x)	# words	# params	Presence	# if present	Presence	# if present
C	61	3	57%	3	84%	5
C++	42	2	54%	3	76%	6
C#	34	2	42%	2	73%	3
Go	42	2	63%	2	85%	6
Java	28	1	37%	2	73%	5
TS	27	1	44%	2	83%	3

Floating-point functions statistics on GitHub

			Conditionals		Function calls		
f(x)	# words	# params	Presence	# if present	Presence	# if present	cos exp sqrt ...
C	61	3	57%	3	84%	5	10%
C++	42	2	54%	3	76%	6	6%
C#	34	2	42%	2	73%	3	6%
Go	42	2	63%	2	85%	6	3%
Java	28	1	37%	2	73%	5	5%
TS	27	1	44%	2	83%	3	2%

Floating-point functions statistics on GitHub

			Conditionals		Function calls		
f(x)	# words	# params	Presence	# if present	Presence	# if present	cos exp sqrt ...
C	61	3	57%	3	84%	5	10%
C++	42	2	54%	3	76%	6	6%
C#	34	2	42%	2	73%	3	6%
Go	42	2	63%	2	85%	6	3%
Java	28	1	37%	2	73%	5	5%
TS	27	1	44%	2	83%	3	2%
FPBench	**23**	**2**	**7%**	**1**	**61%**	**2**	**38%**

Floating-point functions statistics on GitHub

How common is FP code actually?

Most GitHub repos in our dataset (> 62%) contain floating-point code, but usually in only a small share of files.

What do representative FP functions look like?

How common is FP code actually?

Most GitHub repos in our dataset (> 62%) contain floating-point code, but usually in only a small share of files.

What do representative FP functions look like?

Targeting small functions is the right call, but future benchmarks need more conditionals and non-transcendental function calls to be representative.

How can we generate representative benchmarks?

How can we generate representative benchmarks?



Our dataset

How can we generate representative benchmarks?



Our dataset



Self-contained benchmarks

How can we generate representative benchmarks?



Our dataset

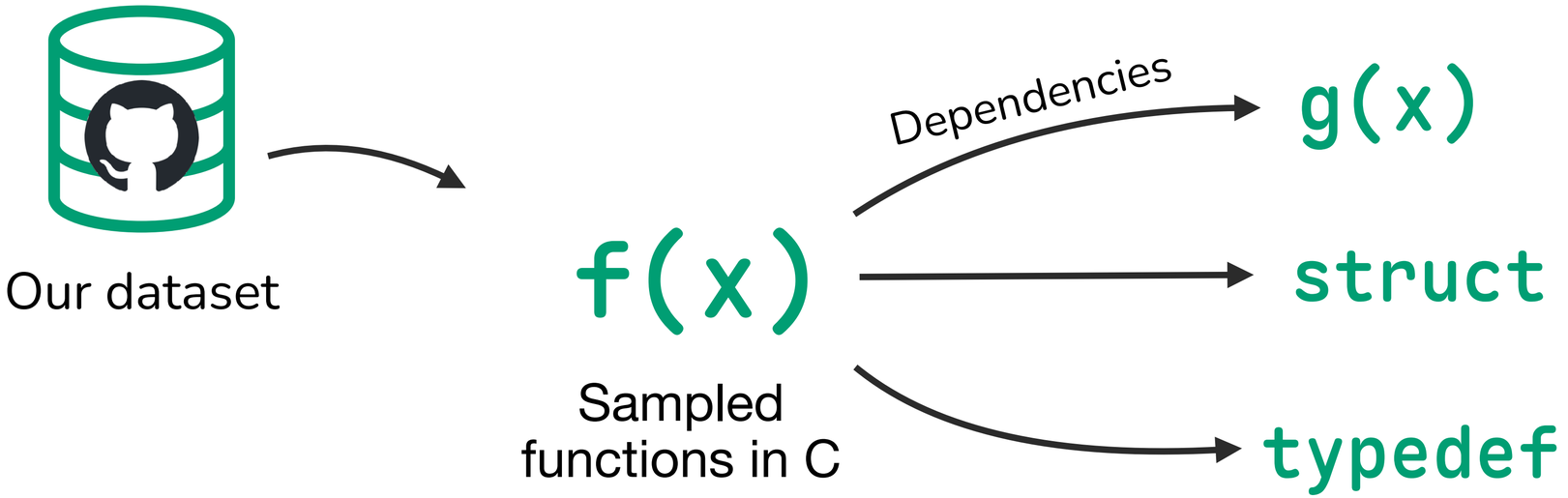
f(x)

Sampled
functions in C



Self-contained
benchmarks

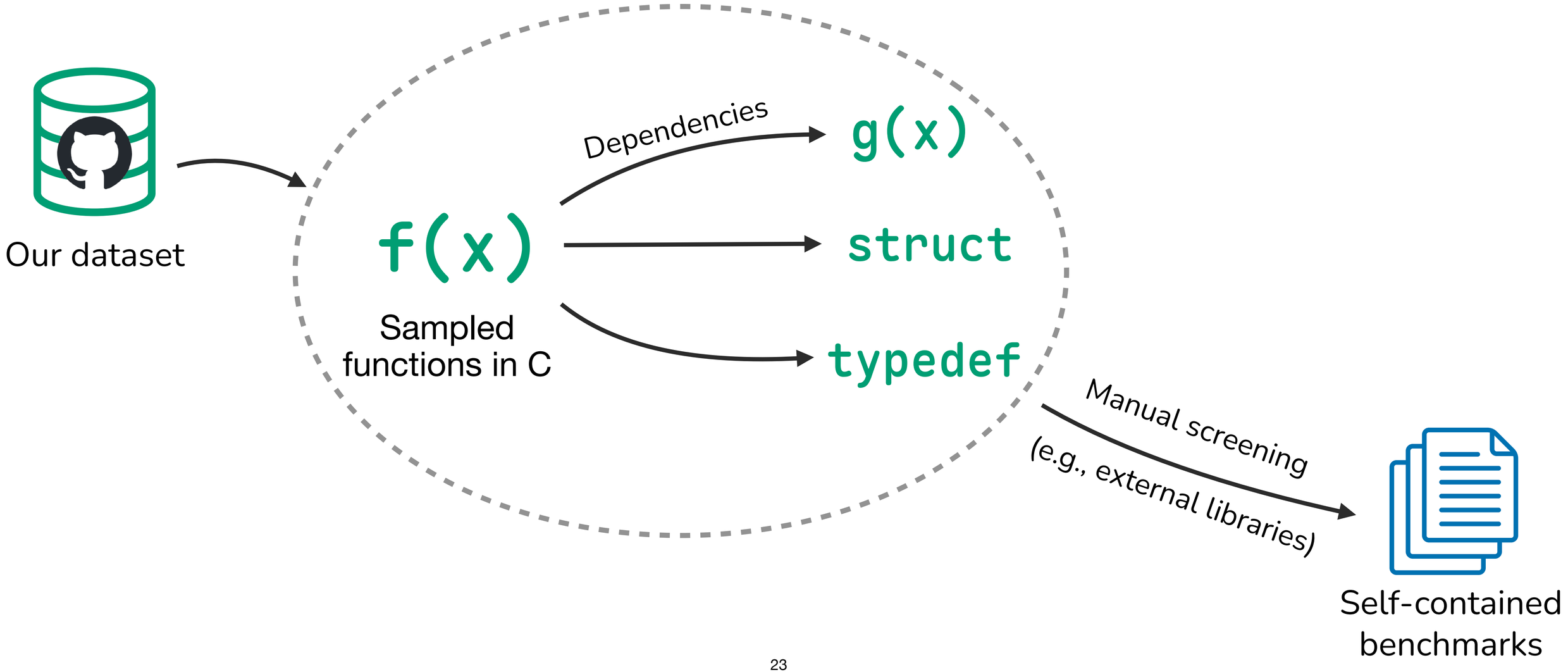
How can we generate representative benchmarks?





Self-contained
benchmarks

How can we generate representative benchmarks?



Verifying Floating-Point Programs in Stainless

Andrea Gilot*✉, Axel Bergström*, and Eva Darulova

Uppsala University
{andrea.gilot,axel.bergstrom,eva.darulova}@it.uu.se

Abstract. We extend the Stainless deductive verifier with floating-point support, providing the first automated verification support for floating-point numbers for a subset of Scala that includes polymorphism, recursion and higher-order functions. We follow the recent approach in the KeY verifier to axiomatise reasoning about mathematical functions, but go further by supporting all functions from Scala's math API, and by verifying the correctness of the axioms against the actual implementation in Stainless itself. We validate Stainless' floating-point support on a new set of benchmarks sampled from real-world code from GitHub, showing that it can verify specifications about, *e.g.*, ranges of output or absence of special values for most supported functions, or produce counter-examples when the specifications do not hold.

Keywords: Floating-point Arithmetic · Transcendental Functions · Deductive Verification

1 Introduction

Floating-point arithmetic is ubiquitous in modern software. While it is often discussed in the context of scientific computing, embedded systems, and machine learning, it is first and foremost pervasive in everyday user code [18]. Yet, verification tools do not always support floating-point reasoning, limiting their use in practice. For example, the absence of floating-point support in Liquid Haskell [40] has been identified as a limitation to its applicability in the real world [39].

This work introduces bit-accurate floating-point reasoning into Stainless [20], a deductive verification framework for Scala programs. Two main challenges in this integration are correctness and performance. To ensure correctness, the implementation must carefully address semantic discrepancies between the JVM and SMT-LIB, and the non-standard behaviour of equality.

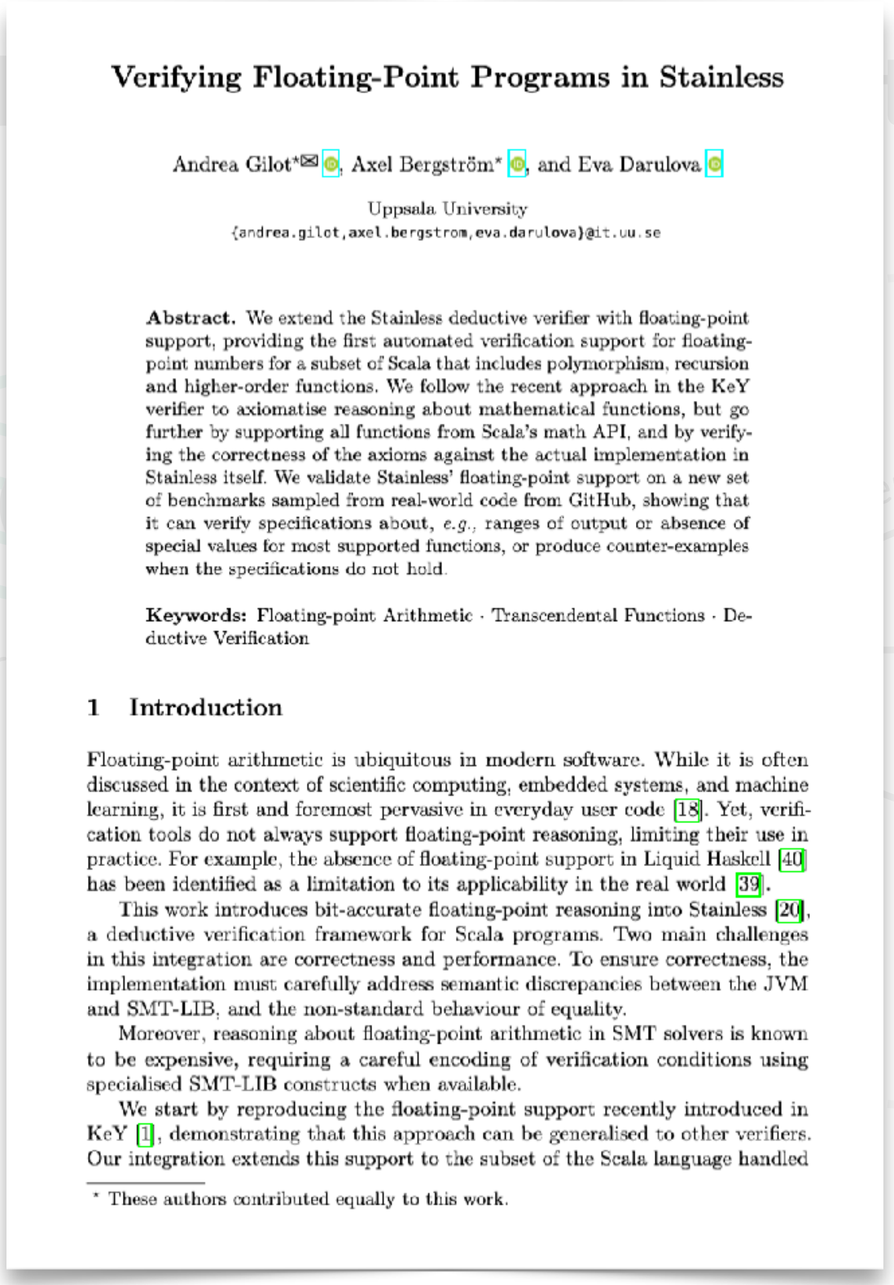
Moreover, reasoning about floating-point arithmetic in SMT solvers is known to be expensive, requiring a careful encoding of verification conditions using specialised SMT-LIB constructs when available.

We start by reproducing the floating-point support recently introduced in KeY [1], demonstrating that this approach can be generalised to other verifiers. Our integration extends this support to the subset of the Scala language handled

* These authors contributed equally to this work.



Self-contained benchmarks



Verifying Floating-Point Programs in Stainless

Andrea Gilot*✉, Axel Bergström*, and Eva Darulova

Uppsala University
{andrea.gilot,axel.bergstrom,eva.darulova}@it.uu.se

Abstract. We extend the Stainless deductive verifier with floating-point support, providing the first automated verification support for floating-point numbers for a subset of Scala that includes polymorphism, recursion and higher-order functions. We follow the recent approach in the KeY verifier to axiomatise reasoning about mathematical functions, but go further by supporting all functions from Scala's math API, and by verifying the correctness of the axioms against the actual implementation in Stainless itself. We validate Stainless' floating-point support on a new set of benchmarks sampled from real-world code from GitHub, showing that it can verify specifications about, *e.g.*, ranges of output or absence of special values for most supported functions, or produce counter-examples when the specifications do not hold.

Keywords: Floating-point Arithmetic · Transcendental Functions · Deductive Verification

1 Introduction

Floating-point arithmetic is ubiquitous in modern software. While it is often discussed in the context of scientific computing, embedded systems, and machine learning, it is first and foremost pervasive in everyday user code [18]. Yet, verification tools do not always support floating-point reasoning, limiting their use in practice. For example, the absence of floating-point support in Liquid Haskell [40] has been identified as a limitation to its applicability in the real world [39].

This work introduces bit-accurate floating-point reasoning into Stainless [20], a deductive verification framework for Scala programs. Two main challenges in this integration are correctness and performance. To ensure correctness, the implementation must carefully address semantic discrepancies between the JVM and SMT-LIB, and the non-standard behaviour of equality.

Moreover, reasoning about floating-point arithmetic in SMT solvers is known to be expensive, requiring a careful encoding of verification conditions using specialised SMT-LIB constructs when available.

We start by reproducing the floating-point support recently introduced in KeY [1], demonstrating that this approach can be generalised to other verifiers. Our integration extends this support to the subset of the Scala language handled

* These authors contributed equally to this work.



Benchmarks with specification



Manual specification



Self-contained benchmarks

Benchmark examples

```
double calSum(double data[], int N, int I) {
    if (I == 0) {
      return data[0];
    }
    double sum = 0.f;
    int idx = 0;
    if (I >= N) {
      idx = N;
    } else {
      idx = I;
    }
    for (int i = 0; i < idx; i++) {
        sum += data[I - i];
    }
    return sum;
}
```

```
void calAR(double PRICE[][4], int count, int N, double AR[]) {
    double hoValue[count];
    double olValue[count];
    for (int i = 0; i < count; i++) {
        hoValue[i] = PRICE[i][0] - PRICE[i][2];
        olValue[i] = PRICE[i][2] - PRICE[i][1];
    }
    for (int i = 0; i < count; i++) {
        double sum1 = calSum(hoValue, N, i);
        double sum2 = calSum(olValue, N, i);
        AR[i] = sum1 / sum2 * 100;
    }
}
```

https://github.com/kayyyuan/YKKline, “K线库”, an iOS stock-chart app

Benchmark examples

```
1 double calSum(double data[], int N, int I) {
2     if (I == 0) {
3       return data[0];
4     }
5     double sum = 0.f;
6     int idx = 0;
7     if (I >= N) {
8       idx = N;
9     } else {
10      idx = I;
11    }
12    for (int i = 0; i < idx; i++) {
13        sum += data[I - i];
14    }
15    return sum;
16 }
```

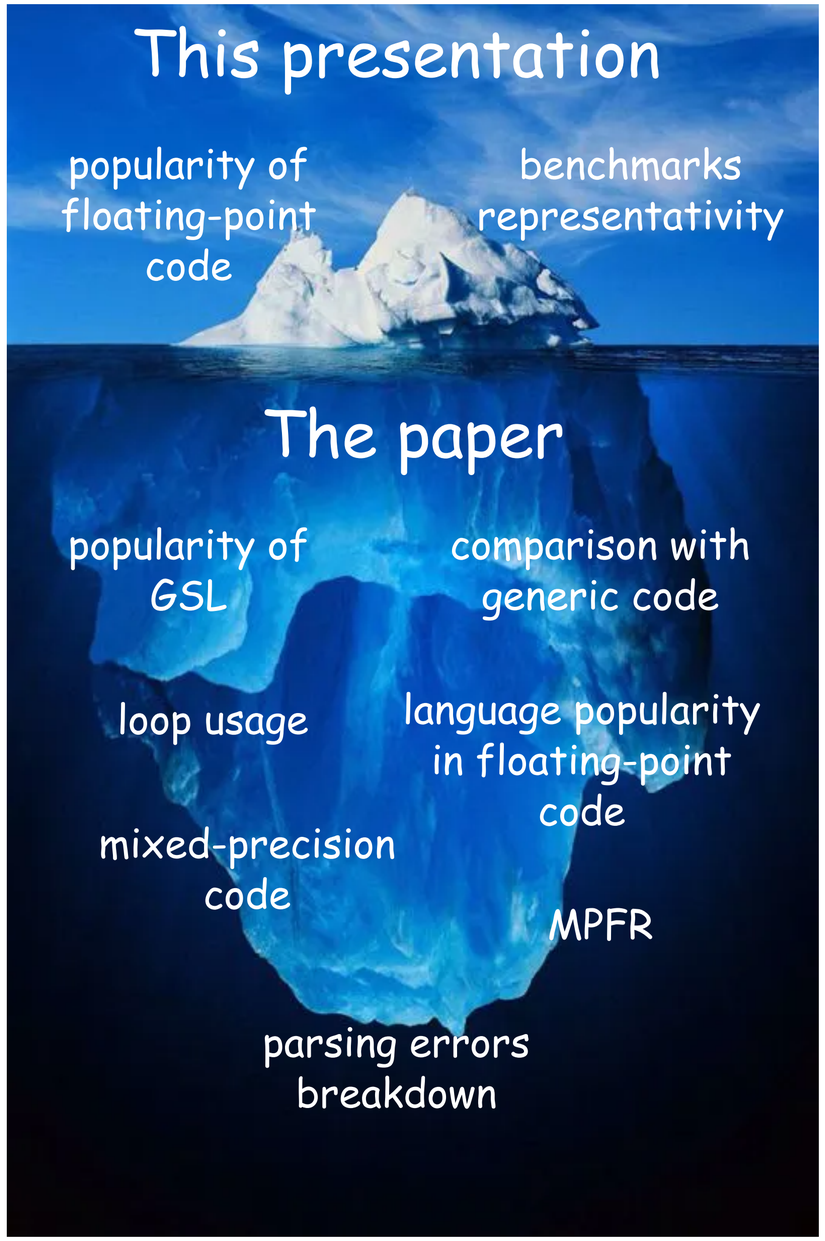
```
17 void calAR(double PRICE[][4], int count, int N, double AR[]) {
18     double hoValue[count];
19     double olValue[count];
20     for (int i = 0; i < count; i++) {
21         hoValue[i] = PRICE[i][0] - PRICE[i][2];
22         olValue[i] = PRICE[i][2] - PRICE[i][1];
23     }
24     for (int i = 0; i < count; i++) {
25         double sum1 = calSum(hoValue, N, i);
26         double sum2 = calSum(olValue, N, i);
27         AR[i] = sum1 / sum2 * 100;
28     }
29 }
```

https://github.com/kayyyuan/YKKline, “K线库”, an iOS stock-chart app

Benchmark examples

```
17 static float quiet_nanf(float x) {
18     uint32_t tmp;
19     memcpy(&tmp, &x, 4);
20     tmp |= 0x7fc00000lu;
21     memcpy(&x, &tmp, 4);
22     return x;
23 }
```

https://github.com/keithw/wabt, “WABT: The WebAssembly Binary Toolkit”





Floating-Point Usage on GitHub: A Large-Scale Study of Statically Typed Languages

ANDREA GILOT, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
EVA DARULOVA, Uppsala University, Sweden

Reasoning about floating-point arithmetic is notoriously hard. While static and dynamic analysis techniques or program repair have made significant progress, more work is still needed to make them relevant to real-world code. On the critical path to that goal is understanding what real-world floating-point code looks like.

To close that knowledge gap, this paper presents the first large-scale empirical study of floating-point arithmetic usage across public GitHub repositories. We focus on statically typed languages to allow our study to scale to millions of repositories. We follow state-of the art mining practices including random sampling and filtering based on only intrinsic properties to avoid bias, and identify floating-point usage by searching for keywords in the source code, and programming language constructs (*e.g.*, loops) by parsing the code. Our evaluation supports the claim often made in papers that floating-point arithmetic is widely used. Comparing statistics such as size and usage of certain constructs and functions, we find that benchmarks used in literature to evaluate automated reasoning techniques for floating-point arithmetic are in certain aspects representative of 'real-world' code, but not in all.

We publish a dataset of 10 million real-world floating-point functions extracted from our study. We demonstrate in a case study how it may be used to identify new floating-point benchmarks and help future techniques for floating-point arithmetic to be designed and evaluated to match actual users' expectations.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Automated static analysis**; • **Information systems** → *Data mining*.

Additional Key Words and Phrases: floating-point arithmetic, large-scale code analysis, repository mining

ACM Reference Format:
Andrea Gilot, Tobias Wrigstad, and Eva Darulova. 2026. Floating-Point Usage on GitHub: A Large-Scale Study of Statically Typed Languages. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 95 (April 2026), 28 pages. https://doi.org/10.1145/3798203

1 Introduction

Numerical software is widely used across different domains such as embedded systems, scientific computing, and machine learning. Such software frequently makes use of floating-point arithmetic to efficiently approximate computations with real numbers. Some approximation of the infinitely precise reals is fundamentally and practically necessary, and introduces rounding errors at most arithmetic operations as well as potentially the special values infinity and Not-a-Number (NaN). Overall this makes reasoning about the correctness of such software unintuitive, error prone and time consuming.

This could be the beginning of a paper on automated reasoning for floating-point programs, *e.g.*, with a static [19, 53] or a dynamic approach [31, 63, 65]. Such an approach would typically be

Authors' Contact Information: Andrea Gilot, Uppsala University, Uppsala, Sweden, andrea.gilot@it.uu.se; Tobias Wrigstad, Uppsala University, Uppsala, Sweden, tobias.wrigstad@it.uu.se; Eva Darulova, Uppsala University, Uppsala, Sweden, eva.darulova@it.uu.se.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART95
https://doi.org/10.1145/3798203



Our dataset, benchmarks & mining tool. Try them in your study!

How common is FP code actually?

Most GitHub repos in our dataset (> 62%) contain floating-point code, but usually in only a small share of files.

What do representative FP functions look like?

Targeting small functions is the right call, but future benchmarks need more conditionals and non-transcendental function calls to be representative.

Pictures attribution

- parcel by wahyu arif nur from Noun Project (CC BY 3.0)
- File by Graphtend from Noun Project (CC BY 3.0)
- open parcel by Soremba from Noun Project (CC BY 3.0)
- dataset by Chenyu Wang from Noun Project (CC BY 3.0)
- Pickaxe by Hanbai from Noun Project (CC BY 3.0)
- hand writing by Juicy Fish from Noun Project (CC BY 3.0)